Chapel Hierarchical Locales

Sung-Eun Choi, David Iten, Elliot Ronaghan, Greg Titus
Chapel Team
Cray Inc.

CHIUW @ PLDI

June 13, 2015

Safe Harbor Statement

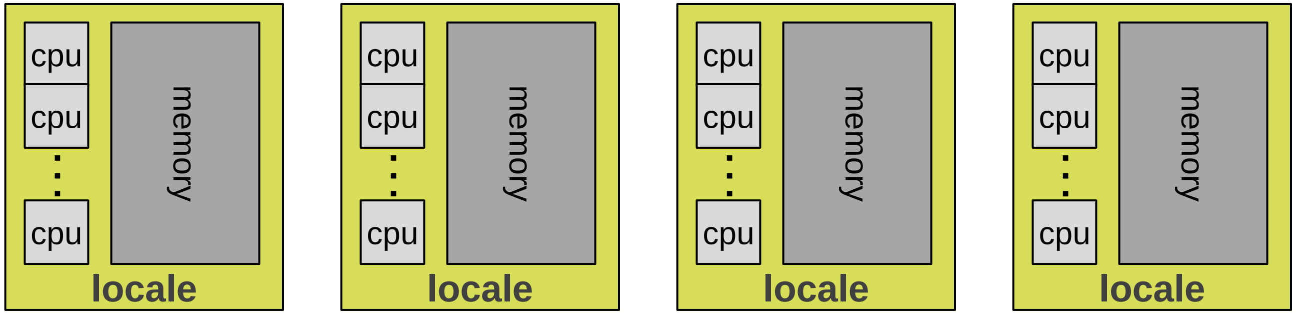
This presentation may contain forward-looking statements that are based on our current expectations. Forward looking statements may include statements about our financial guidance and expected operating results, our opportunities and future potential, our product development and new product introduction plans, our ability to expand and penetrate our addressable markets and other statements that are not historical facts. These statements are only predictions and actual results may materially vary from those projected. Please refer to Cray's documents filed with the SEC from time to time concerning factors that could affect the Company and these forward-looking statements.

Outline

- **The problem: architecture and how to express it**
- **The solution: *hierarchical locales***
- **Locality during compilation**
- **Status and plans**

Chapel's Architecture Model Was So Simple

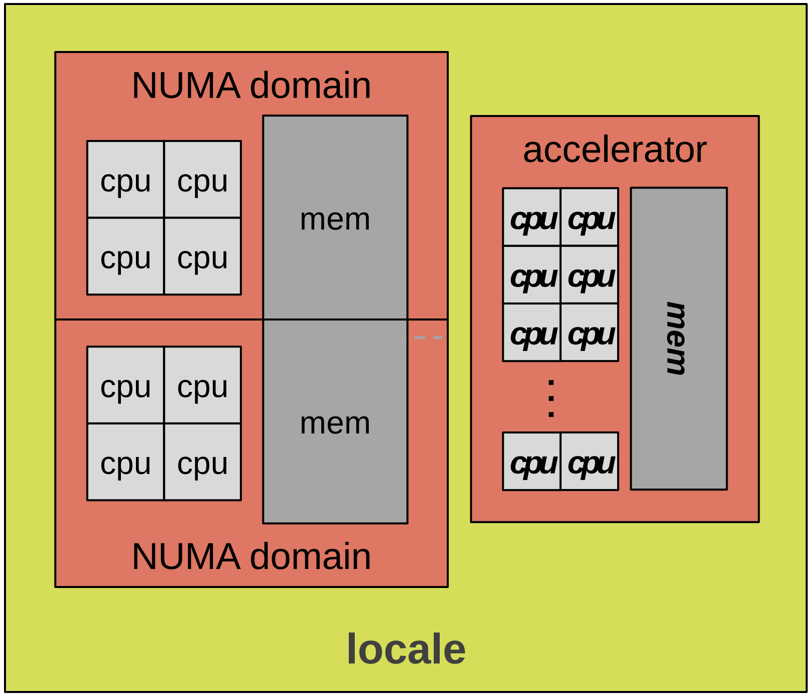
- **Traditionally, Chapel supported only a 1D array of locales**
  - Users could reshape/slice to suit their computation's needs



- **Apart from queries, no further visibility into locales**
- **Supports top level inter-node locality well**
  - Assumes target compiler, runtime, OS, HW can handle intra-locale concerns

Chapel's Architecture Model Was *Too* Simple

- **(HPC) architectures are varied and evolving rapidly**
  - Intra-node architecture becoming complex and important
  - Hierarchy (example: NUMA)
  - Heterogeneity (example: GPUs)



- **Performance requires using *all* architecture effectively**
  - But Chapel had no mechanism to refer to intra-node details

- **Need access to NUMA domains, CPUs, memories, etc.**

What are the Requirements?

- **Have just 3 architecture-dependent classes of operations:**
  - **Memory management (allocate, free, etc.)**
  - **Task support (initiate, move, etc.)**
  - **Communication**
  - Helpful: do not need very many operations from each class

- **Solution must be approachable, adaptable, flexible**
  - Knowing Chapel + architecture and being motivated should be enough
    - Should not require "magic" and/or Chapel core team help
  - Must support experimentation and prototyping

Outline

- ✓ **The problem: architecture and how to express it**
- ➢ **The solution: *hierarchical locales***
- **Locality during compilation**
- **Status and plans**

Solution: Chapel Hierarchical Locales

- **Standardized class describes CPU+mem architecture**

```
class LocaleModel { … }
```

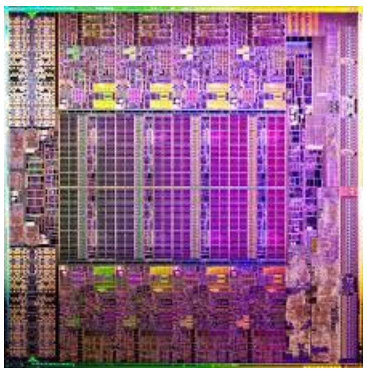
- **Composable, to reflect hierarchy**

- **Has required interface, referenced by generated code**
  - Memory management
  - Task support
  - Communication operations
  - Hierarchical relatives (parents, siblings, children)

- **May be implemented however desired**
  - Typically in terms of other LocaleModel instances or runtime calls

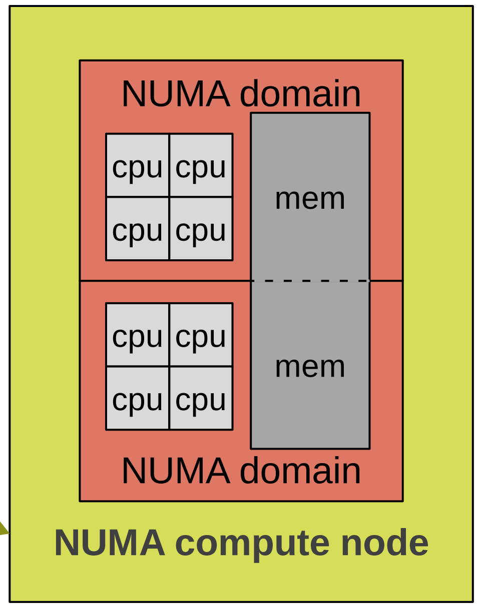
Example: The Predefined numa Locale Model

physical



http://www1.pcmag.com/media/images/337192-intel-xeon-e5-chip.jpg?thumb=y

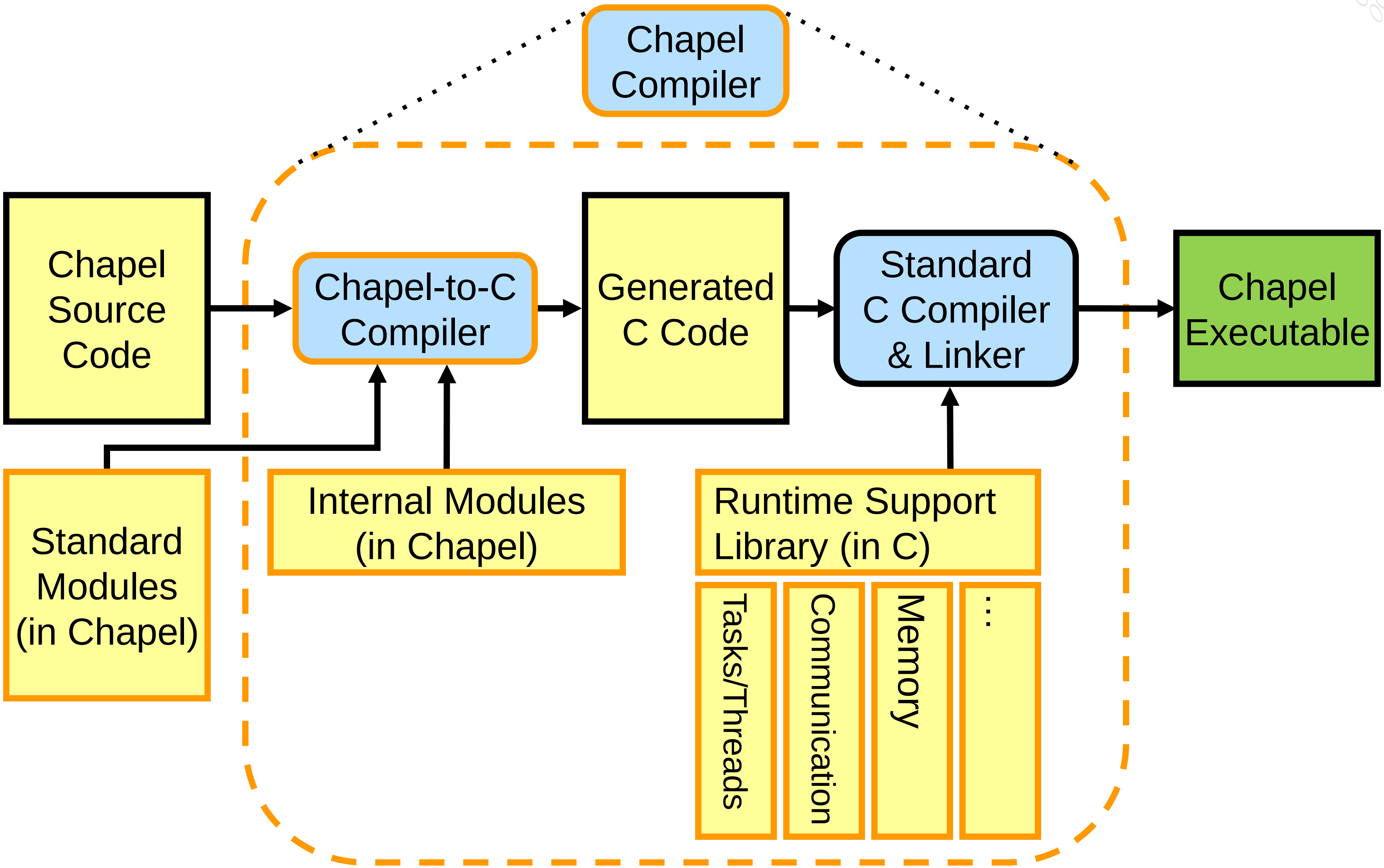
conceptual



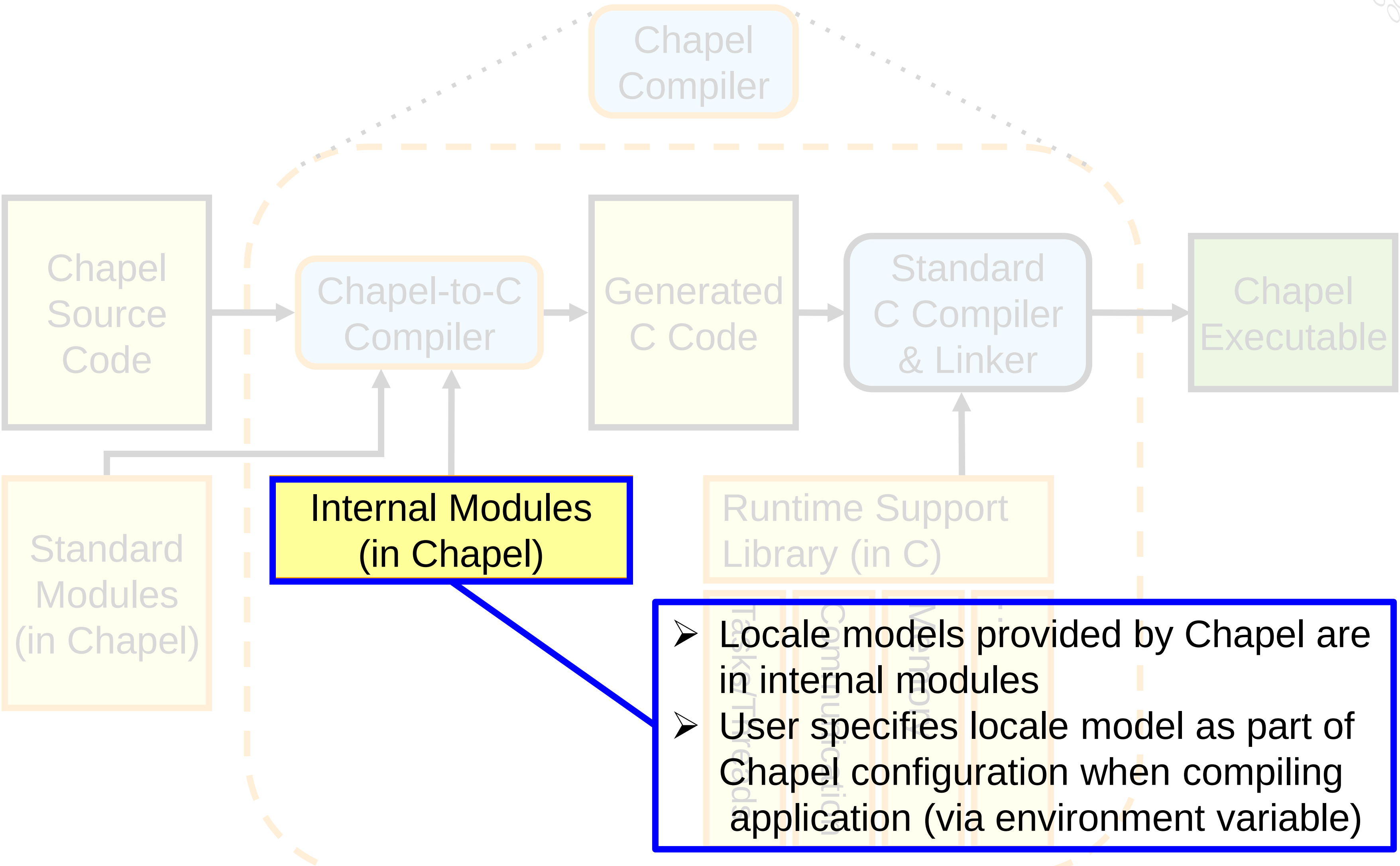
$CHPL_HOME/modules/…/numa/LocaleModel.chpl

```
class NumaDomain : AbstractLocaleModel {
  const sid: chpl_sublocID_t;
}

// The node model
class LocaleModel : AbstractLocaleModel {
  const numSublocales: int;
  var childSpace: domain(1);
  var childLocales: [childSpace] NumaDomain;
}

// support for memory management
proc chpl_here_alloc(size:int, md:int(16)) { … }

// support for "on" statements
proc chpl_executeOn
      (loc: chpl_localeID_t, // target locale
       fn: int,              // on-body func idx
       args: c_void_ptr,     // func args
       args_size: int(32)    // args size
      ) { … }

// support for tasking stmts: begin, cobegin, coforall
proc chpl_taskListAddCoStmt
      (subloc_id: int,         // target subloc
       fn: int,                // body func idx
       args: c_void_ptr,       // func args
       ref tlist: _task_list,  // task list
       tlist_node_id: int      // task list owner
      ) { … }
```

Where Predefined Locale Models Live



Where Predefined Locale Models Live



- Locale models provided by Chapel are in internal modules
- User specifies locale model as part of Chapel configuration when compiling application (via environment variable)

Hierarchical Locales Create a New Chapel Role

- **Application programmer: work on applications**
  - Express solutions in a natural way
  - Use forall statements to expose data parallelism
  - Use domain maps to inform Chapel about locality and affinity

Hierarchical Locales Create a New Chapel Role

- **Application programmer: work on applications**
  - Express solutions in a natural way
  - Use forall statements to expose data parallelism
  - Use domain maps to inform Chapel about locality and affinity

- **Domain map specialist: work on locality**
  - In a general or conceptual way, not an architecture-specific one

Hierarchical Locales Create a New Chapel Role

- **Application programmer: work on applications**
  - Express solutions in a natural way
  - Use forall statements to expose data parallelism
  - Use domain maps to inform Chapel about locality and affinity

- **Domain map specialist: work on locality**
  - In a general or conceptual way, not an architecture-specific one

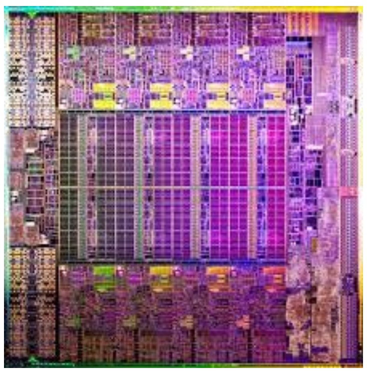
★ **Architecture modeler: work on architectural mappings**
  - Describe architectural hierarchy
  - Implement functional interfaces at various levels

Outline

- ✓ **The problem: architecture and how to express it**
- ✓ **The solution: *hierarchical locales***
- ➢ **Locality during compilation**
- **Status and plans**

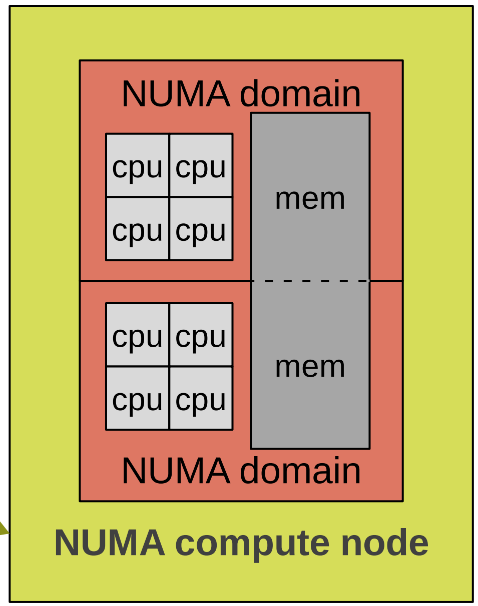
Context: Using Predefined numa Locale Model

physical



http://www1.pcmag.com/media/images/337192-intel-xeon-e5-chip.jpg?thumb=y

conceptual

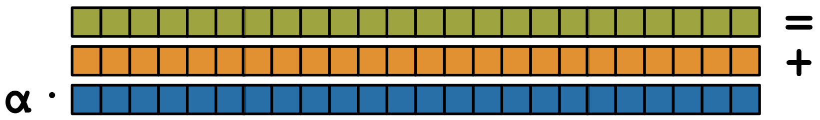


$CHPL_HOME/modules/…/numa/LocaleModel.chpl

```
class NumaDomain : AbstractLocaleModel {
  const sid: chpl_sublocID_t;
}

// The node model
class LocaleModel : AbstractLocaleModel {
  const numSublocales: int;
  var childSpace: domain(1);
  var childLocales: [childSpace] NumaDomain;
}

// support for memory management
proc chpl_here_alloc(size:int, md:int(16)) { … }

// support for "on" statements
proc chpl_executeOn
     (loc: chpl_localeID_t, // target locale
      fn: int,              // on-body func idx
      args: c_void_ptr,     // func args
      args_size: int(32)    // args size
     ) { … }

// support for tasking stmts: begin, cobegin, coforall
proc chpl_taskListAddCoStmt
     (subloc_id: int,        // target subloc
      fn: int,               // body func idx
      args: c_void_ptr,      // func args
      ref tlist: _task_list, // task list
      tlist_node_id: int     // task list owner
     ) { … }
```

The Application, as Architecture-free Code

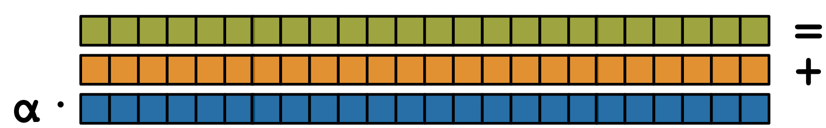
```
// Stream Triad
config const m = 1000,
              alpha = 3.0;
const ProblemSpace = {1..m} dmapped Block(…);
var A, B, C: [ProblemSpace] real;
B = 2.0;
C = 3.0;
A = B + alpha * C;
```



The Application, as Architecture-free Code

Express parallelism abstractly, without referring to physical architecture

```
// Stream Triad
config const m = 1000,
             alpha = 3.0;
const ProblemSpace = {1..m} dmapped Block(…);
var A, B, C: [ProblemSpace] real;
B = 2.0;
C = 3.0;
A = B + alpha * C;
```

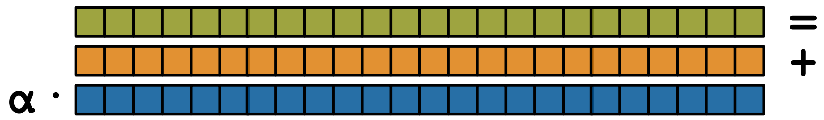


The Application, as Architecture-free Code

Specify domain map in application code

Express parallelism abstractly, without referring to physical architecture

```
// Stream Triad
config const m = 1000,
             alpha = 3.0;
const ProblemSpace = {1..m} dmapped Block(…);
var A, B, C: [ProblemSpace] real;
B = 2.0;
C = 3.0;
A = B + alpha * C;
```



Locality & Affinity in the Domain Map

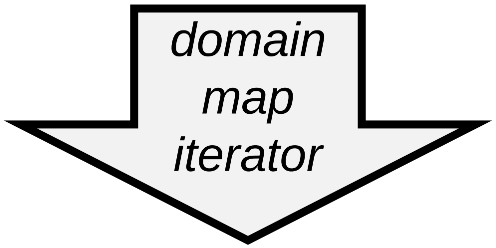
```
// Block domain map
class Block: BaseDist {
  var targetLocDom: domain(rank);
  var targetLocales: [targetLocDom] locale;
  var dataParTasksPerLocale: int;
  var dataParIgnoreRunningTasks: bool;
  var dataParMinGranularity: int;
}
…
iter these(param tag: iterKind,
       tasksPerLocale = dataParTasksPerLocale,
       ignoreRunning = dataParIgnoreRunningTasks,
       minIndicesPerTask = dataParMinGranularity)
{
  const numSublocs = here.getChildCount();
  if locModelHasSublocs && numSublocs != 0 {
    … _computeChunkStuff(min(numSublocs,
                          here.maxTaskPar),
                         ignoreRunning,
                         minIndicesPerTask,
                         ranges);
    …
  }
}
```

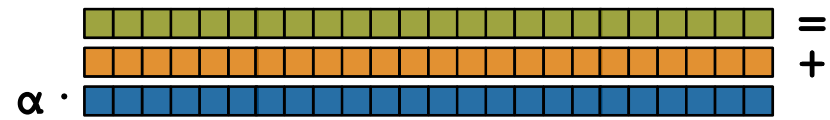
Domain map:

- Describes distribution of indices (block, cyclic, etc.)
- Ties together locality, affinity, parallelism via iterators for forall-stmts
- Has a standardized interface, referenced by compiler-generated code
- Can interrogate locale model to learn about resources
- Is typically coded by a specialist

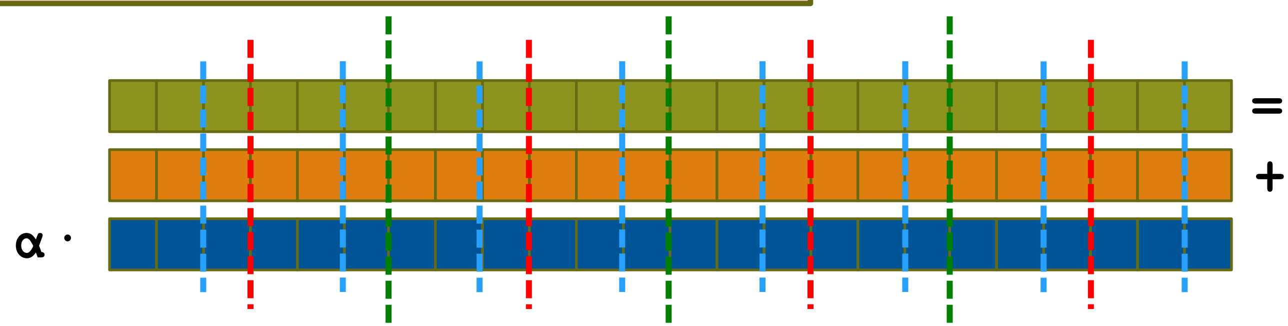
The Application, Translated by the Domain Map

```
const ProblemSpace = {1..m} dmapped Block(…);
var A, B, C: [ProblemSpace] real;
A = B + alpha * C;
```






```
coforall loc in targetLocales do on loc {
  coforall subloc in loc.getChildren() do on subloc {
    coforall tid in here.numCores {
      for (a,b,c) in zip(A,B,C) do a = b + alpha * c;
    }
  }
}
```



… Translated Again, by the Chapel Compiler

```
coforall loc in targetLocales do on loc {
  coforall subloc in loc.getChildren() do on subloc {
    coforall tid in here.numCores {
      for (a,b,c) in zip(A,B,C) do a = b + alpha * c;
    }
  }
}
```

Chapel code

Chapel compiler

```
void main(…) {
  chpl_taskListAddCoStmt(fn_for_outer_coforall_stmt);
}
void fn_for_outer_coforall_stmt(…) {
  chpl_executeOn(loc, fn_for_on_stmt);
}
void fn_for_on_stmt(…) {
  chpl_taskListAddCoStmt(fn_for_middle_coforall_stmt);
}
void fn_for_middle_coforall_stmt(…) {
  chpl_taskListAddCoStmt(fn_for_inner_coforall_stmt);
}
void fn_for_inner_coforall_stmt(…) {
  for (…) { a[i] = b[i] + alpha * c[i]; }
}
```

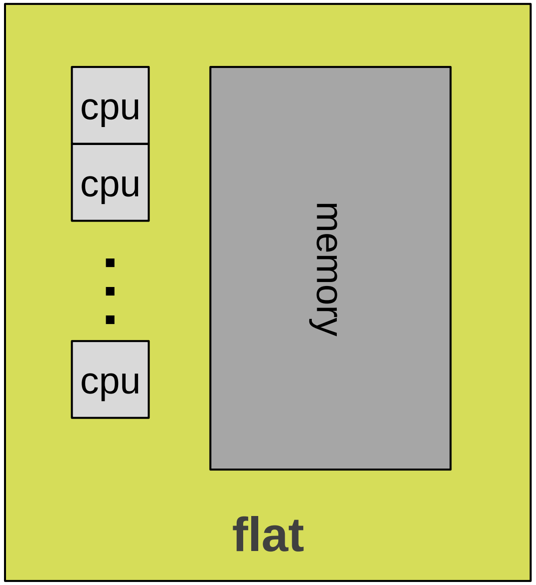
C code

Outline

- ✓ **The problem: architecture and how to express it**
- ✓ **The solution: *hierarchical locales***
- ✓ **Locality during compilation**
- ➢ **Status and plans**

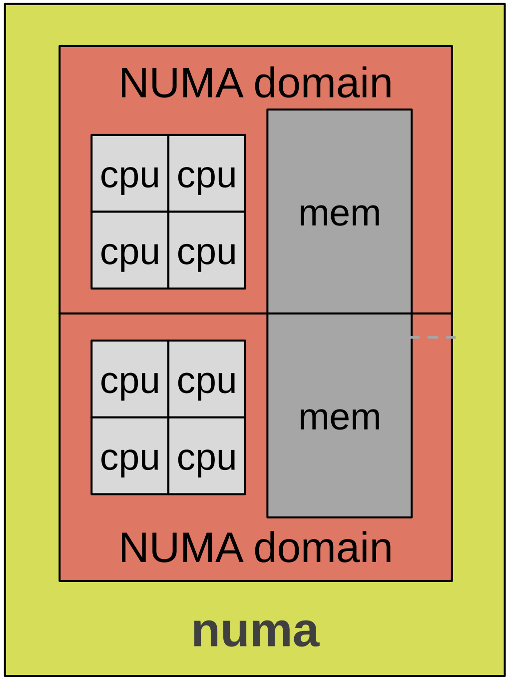
Today's Locale Models: flat

- Direct replacement for the old compiler-implemented model
- Same performance as old compiler-based architecture support
- Default in all cases



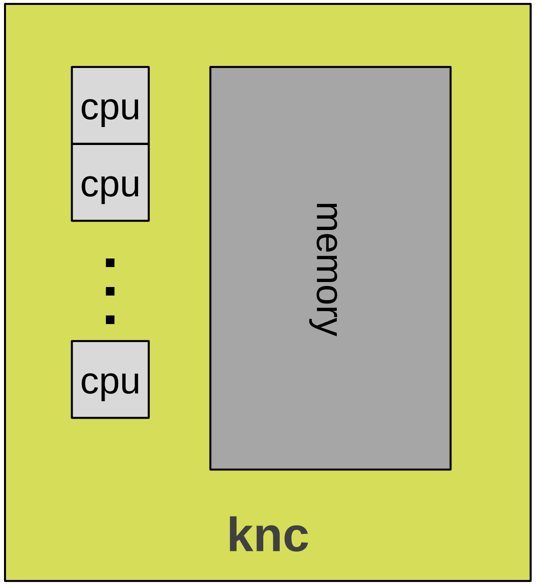
Today's Locale Models: numa

- **Functional**
  - Tasks follow memory affinity properly
- **Performance needs improvement**
  - Auto-init puts all mem on numa node 0
  - Working on memory locality
    - Auto-init improvements
    - Low-level memory management
  - Working on execution locality
    - Improving numa handling in Qthreads
- **Aiming at fall 2015 release**



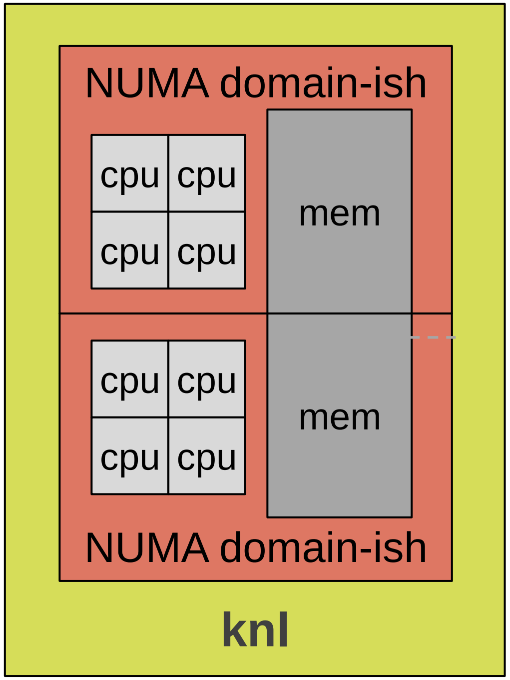
Tomorrow's Locale Models: "real" knc

- Current Chapel Intel Xeon Phi KNC support uses "flat"
- Duplicate and tune for KNC-specific properties (breadth, e.g.)



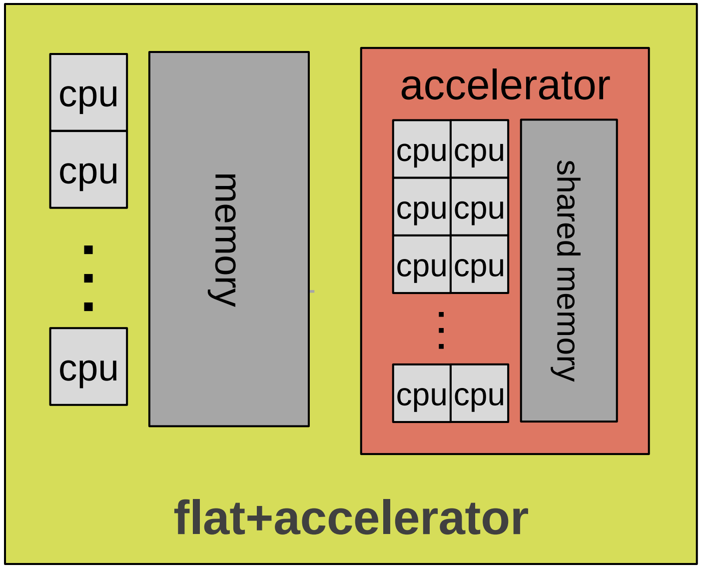
Tomorrow's Locale Models: knl

- **Intel Xeon Phi KNL would be an elaboration of numa**
  - Similar to flat → knc
- **Working on:**
  - Access to high bandwidth memory
    - Additional sub-locale?
    - Specialized memory management
  - KNL-specific Qthreads improvements
- **Aiming at spring 2016 release**



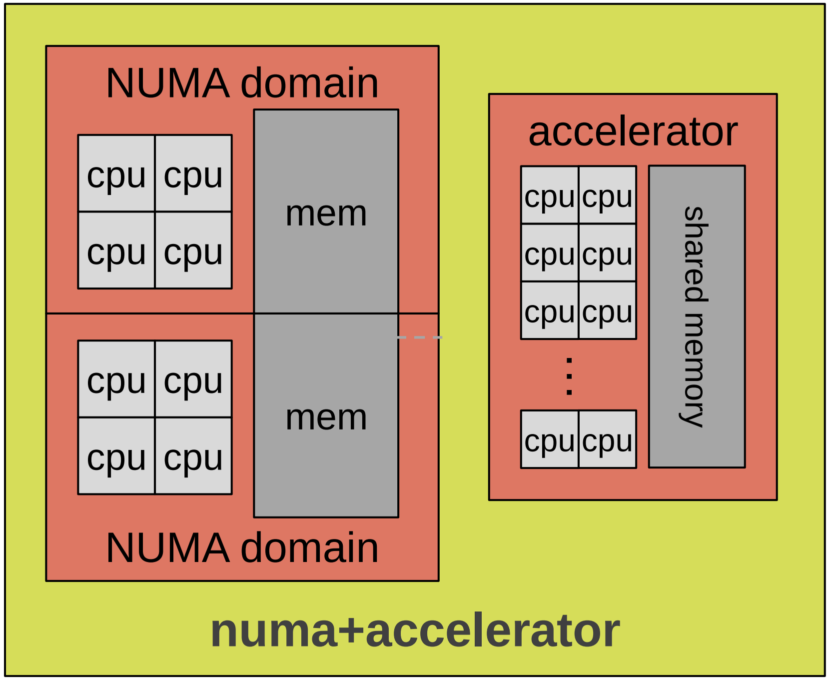
Tomorrow's Locale Models: accelerator

- Challenge: processor heterogeneity



Tomorrow's Locale Models: numa+accelerator

- **Challenge: hierarchy *and* heterogeneity**
- **Good composability test**



Summary

- **Hierarchical Locales feature helps "future proof" Chapel**

- **Enables separation of concerns**
  - **Application programmers** are freed from architecture concerns
  - **Domain map programmers** are freed from architecture concerns
  - **Compiler** is freed from architecture concerns
  - Even the **Chapel language** is freed from architectural concerns

- **Puts Chapel architectural policy in the hands of those most qualified to deal with it: architecture experts**

Legal Disclaimer

Information in this document is provided in connection with Cray Inc. products. No license, express or implied, to any intellectual property rights is granted by this document.

Cray Inc. may make changes to specifications and product descriptions at any time, without notice.

All products, dates and figures specified are preliminary based on current expectations, and are subject to change without notice.

Cray hardware and software products may contain design defects or errors known as errata, which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Cray uses codenames internally to identify products that are in development and not yet publically announced for release. Customers and other third parties are not authorized by Cray Inc. to use codenames in advertising, promotion or marketing and any use of Cray Inc. internal codenames is at the sole risk of the user.

Performance tests and ratings are measured using specific systems and/or components and reflect the approximate performance of Cray Inc. products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance.

The following are trademarks of Cray Inc. and are registered in the United States and other countries: CRAY and design, SONEXION, URIKA, and YARCDATA. The following are trademarks of Cray Inc.: ACE, APPRENTICE2, CHAPEL, CLUSTER CONNECT, CRAYPAT, CRAYPORT, ECOPHLEX, LIBSCI, NODEKARE, THREADSTORM. The following system family marks, and associated model number marks, are trademarks of Cray Inc.: CS, CX, XC, XE, XK, XMT, and XT. The registered trademark LINUX is used pursuant to a sublicense from LMI, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis. Other trademarks used in this document are the property of their respective owners.